**PHASE 5: Smart Parking System Documentation**

**Project Title: SMART PARKING**

**Project ID:**  proj\_223739\_Team\_5

**College Code:** 6208

**College:** Gnanamani College of Technology

**Branch:** B.Tech-Information Technology

**Year:** IIIrd year

**Team Members & ID:**

ABISHEK R - (au620821205002)

SANJAY S - (au620821205047)

PRIYADHARSAN B -(au620821205303)

KARTHICKRAJA M -(au620821205025)

**SMART PARKING**

**Definition:**

An IoT-based parking system is a centralized management that enables drivers to search for and reserve a parking spot remotely through their smartphones. It offers a convenient arrangement for drivers to park their cars when they are looking to avoid potential traffic congestion.

The system’s hardware sensors detect available slots and communicate the information to the drivers in that area in real-time. IoT technology ensures that they do not have to worry about finding an available space again – allowing them to travel conveniently.

**Components Required:**

* Arduino UNO
* Arduino Cable
* LCD Display
* Servo Motor
* IR Sensor
* Bread Board
* Jumpers
* USB Cables

**PHASE 5: Smart Parking System Documentation**

**Project Objectives**

The Smart Parking project aims to address urban parking challenges by providing real-time parking availability information to drivers. The objectives include:

**Efficient Space Utilization:** Ensure that parking spaces are used optimally, reducing wasted space and increasing parking availability.

**Real-Time Availability Information:** Provide drivers with accurate, up-to-the-minute information on available parking spots, making it easier for them to find parking quickly.

**Reduced Congestion:** Alleviate traffic congestion by guiding drivers to open parking spaces efficiently, reducing the time spent searching for parking.

**Enhanced User Experience:** Develop a user-friendly mobile application or interface that allows drivers to easily access parking information, make reservations, and pay for parking.

**Environmental Sustainability:** Help reduce emissions and lower fuel consumption by minimizing the time vehicles spend idling or circling to find parking.

**Data-Driven Insights:** Collect and analyze data on parking patterns, usage, and peak demand to inform urban planning and transportation management.

**Security and Safety**: Implement security features in parking facilities, such as surveillance cameras and emergency assistance options, to enhance the safety of both vehicles and pedestrians.

**Revenue Generation:** Create opportunities for parking operators to maximize revenue through dynamic pricing models, reservation systems, and improved parking services.

**Integration with Navigation Systems:** Seamlessly integrate parking availability information with popular navigation and mapping applications to offer drivers a complete transportation solution.

**Accessibility and Inclusivity:** Ensure that parking facilities are accessible to individuals with disabilities, promoting inclusivity and compliance with accessibility standards.

**Maintenance and Monitoring:** Implement IoT sensors to monitor the condition of parking infrastructure, enabling proactive maintenance and quick responses to issues.

**Urban Planning Support:** Provide urban planners with valuable data and insights to support decisions related to parking infrastructure development, zoning, and transportation planning.

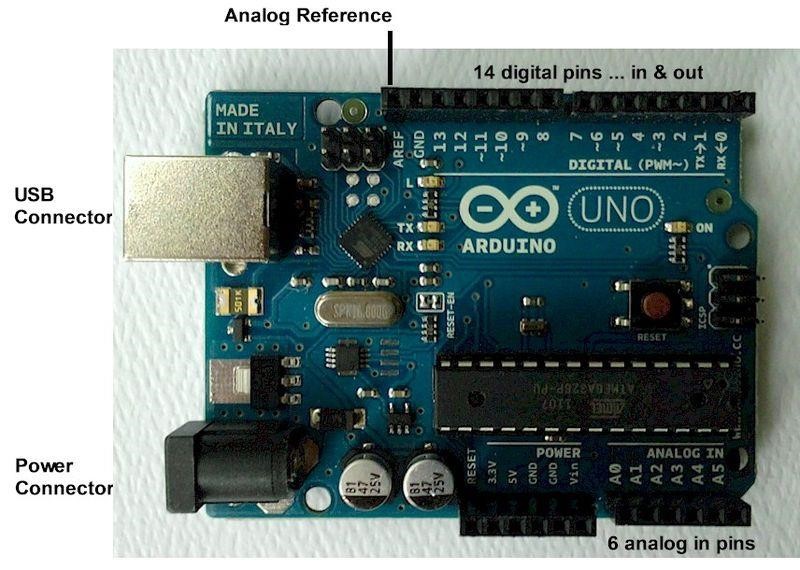
**IoT Sensor Setup**

**1.Hardware**

* **Arduino UNO:**

The Arduino Uno is a microcontroller board. It is used ATmega328p. Arduino Uno have 14 input/output pins and output pin (PMW outputs as of which 6 can be used) and 6 analogy inputs. The Arduino Uno circuit acts as an interface between the software part and the hardware part of the project. Arduino board is one type of microcontroller. It is able to read input like light sensor, detect motion and gives an appropriate output on it. Arduino works like a brain so here we can store.

The microcontrollers are typically programmed using a dialect of features from the programming languages [C](https://en.wikipedia.org/wiki/C_%28programming_language%29) and [C++.](https://en.wikipedia.org/wiki/C%2B%2B) In addition to using traditional compiler tool chains, the Arduino project provides an [integrated development environment](https://en.wikipedia.org/wiki/Integrated_development_environment) (IDE) based on the [Processing](https://en.wikipedia.org/wiki/Processing_%28programming_language%29) language project.



**Fig:** Arduino UNO

* **Arduino Cable**

An Arduino typically only requires a USB cable to be used as a power source and to connect to a computer for programming. A USB Type B cable can be used for the most popular Arduino model (Uno), however other Arduino boards may require Type B-mini or micro connectors.

**Fig:** Arduino Cable

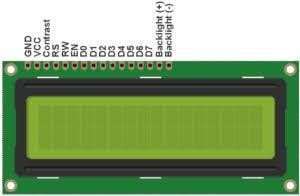


* **LCD Display**

A liquid-crystal display (LCD) is a [flat-panel display](https://en.wikipedia.org/wiki/Flat_panel_display) or other [electronically modulated optical device](https://en.wikipedia.org/wiki/Electro-optic_modulator) that uses the light-modulating properties of [liquid crystals.](https://en.wikipedia.org/wiki/Liquid_crystal) Liquid crystals do not emit light directly, instead using a [backlight](https://en.wikipedia.org/wiki/Backlight) or [reflector](https://en.wikipedia.org/wiki/Reflector_%28photography%29) to produce images in color or [monochrome](https://en.wikipedia.org/wiki/Monochrome)[.[1]](https://en.wikipedia.org/wiki/Liquid-crystal_display#cite_note-1) LCDs are available to display arbitrary images (as in a general-purpose computer display) or fixed images with low information content, which can be displayed or hidden, such as preset words, digits, and [seven-segment displays.](https://en.wikipedia.org/wiki/Seven-segment_display)

**Pin Description:**

|  |  |  |
| --- | --- | --- |
| **Pin No** | **Function** | **Name** |
| 1 | Ground (0V) | Ground |
| 2 | Supply voltage; 5V (4.7V – 5.3V) | Vcc |
| 3 | Contrast adjustment; through a variable resistor | VEE |
| 4 | Selects command register when low; and data register when high | Register Select |
| 5 | Low to write to the register; High to read from the register | Read/write |
| 6 | Sends data to data pins when a high to low pulse is given | Enable |
| 7 | 8-bit data pins | DB0 |
| 8 | DB1 |
| 9 | DB2 |
| 10 | DB3 |
| 11 | DB4 |
| 12 | DB5 |
| 13 | DB6 |
| 14 | DB7 |
| 15 | Backlight VCC (5V) | Led+ |
| 16 | Backlight Ground (0V) | Led- |



**Fig:** 16X2 LCD Display

* **Bread Board**

A breadboard, solderless breadboard, or protoboard is a construction base used to build semi-permanent prototypes of electronic circuits. Unlike a perfboard or stripboard, breadboards do not require soldering or destruction of tracks and are hence reusable.



* **Raspberry Pi**

The Raspberry Pi is a low cost, **credit-card sized computer** that plugs into a computer monitor or TV, and uses a standard keyboard and mouse. It is a capable little device that enables people of all ages to explore computing, and to learn how to program in languages like Scratch and Python. It’s capable of doing everything you’d expect a desktop computer to do, from browsing the internet and playing high-definition video, to making spreadsheets, word-processing, and playing games.



**Fig:** Rasbperry pi

**2.Sensor**

* **IR Sensor**

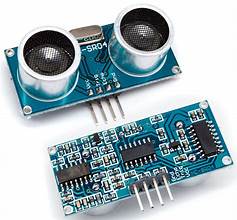
An IR sensor is an electronic device that emits to sense some aspects of the surroundings. An IR sensor can measure the heat of an object as well as detects the motion. These types of sensors measure only IR radiation rather than emitting it, that is called as a passive IR sensor.

**Fig:** IR Sensor



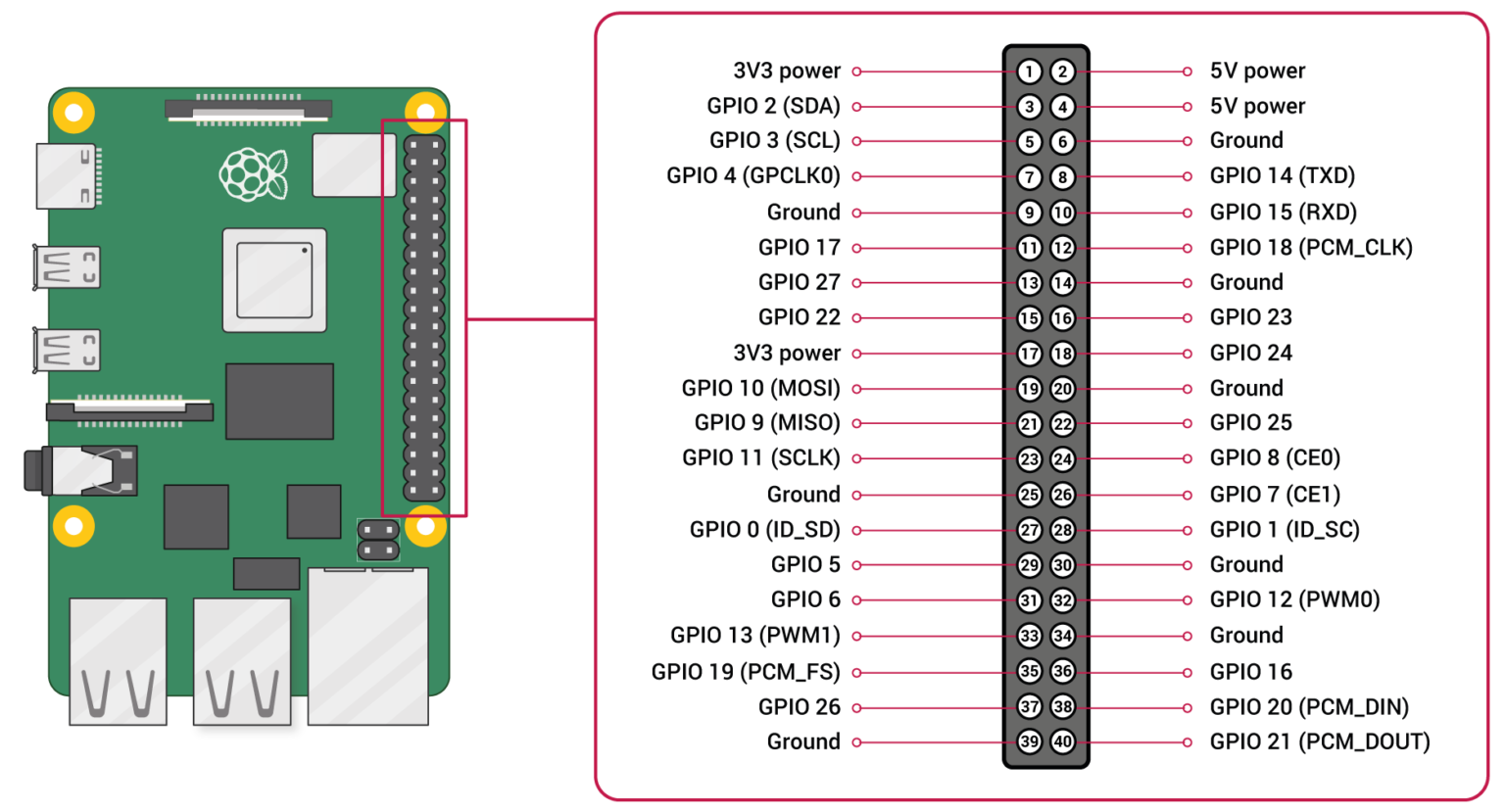
* **HC-SR04 ultrasonic**

The HC-SR04 ultrasonic sensor uses sonar to determine the distance to an object. This sensor reads from 2cm to 400cm (0.8inch to 157inch) with an accuracy of 0.3cm (0.1inches), which is good for most hobbyist projects. In addition, this particular module comes with ultrasonic transmitter and receiver modules.



**Fig:** HC-SR04 ultrasonic sensor

**Raspberry Pi Pinout:**

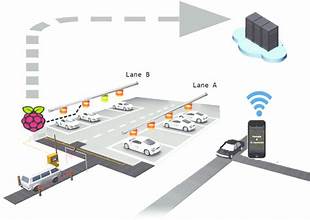


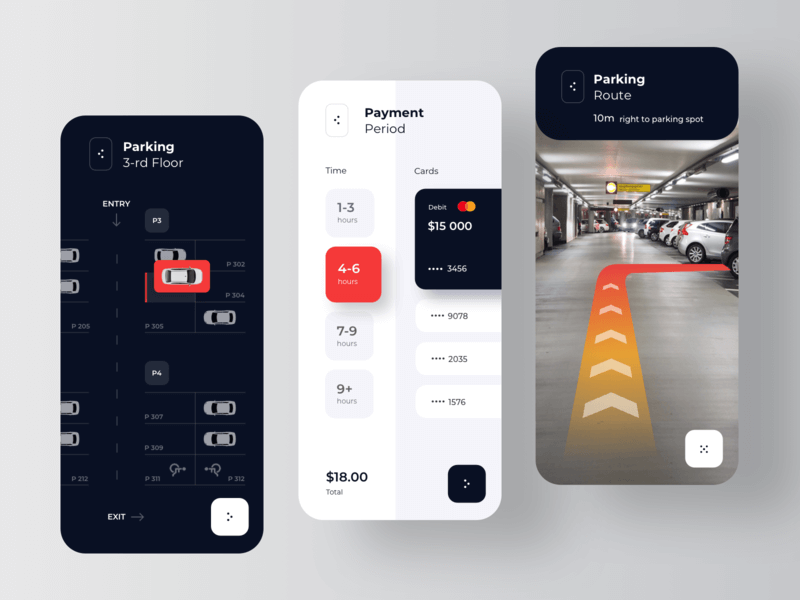
**Mobile App Development:**

Python and Dropbox (as Cloud) - Raspberry is the link between the sensor and theCloud, and the scripts written in Python realize that. The script activates pins and sendsa high-frequency signal of one nanosecond and measures the time between sending andreceiving the signal. The distance is obtained from the multiplication of the interval andthe speed of the ultrasonic signal (34300cm/s) divided by two. If the sensor is mountedon the ceiling, we can determine the approximate distance required for the parking spaceto be occupied. After that, the data is sent to the Cloud because Raspberry cannot process large amounts of data, while the Cloud has data for the entire network and not just for one garage.

**Diagram:**







**Raspberry Pi Integration:**

This module focuses on processing and storing data about parking space availability using the Raspberry Pi.

**Data processing algorithms and data storage :** Algorithms for processing and storing data about parking space availability. You could use database systems like MySQL, or NoSQL databases like MongoDB, and write code for data retrieval and storage. User Interface (Mobile App/Web App) Development

Code:

import RPi.GPIO as GPIO

import time

import sqlite3

# Initialize GPIO

GPIO.setmode(GPIO.BCM)

sensor\_pin = 17 # Change this to your sensor's GPIO pin

# Initialize the database

conn = sqlite3.connect("smart\_parking.db")

cursor = conn.cursor()

# Create a table to store parking spot data

cursor.execute('''CREATE TABLE IF NOT EXISTS parking\_data (

spot\_id INTEGER PRIMARY KEY,

status INTEGER,

timestamp DATETIME DEFAULT CURRENT\_TIMESTAMP)''')

conn.commit()

try:

GPIO.setup(sensor\_pin, GPIO.IN)

while True:

# Read sensor data (replace this with your actual sensor reading logic)

sensor\_data = GPIO.input(sensor\_pin)

# Process and store the data in the database

cursor.execute("INSERT INTO parking\_data (status) VALUES (?)", (sensor\_data,))

conn.commit()

# Print for debugging (you can remove this in a production system)

print(f"Sensor Data: {sensor\_data}")

time.sleep(5) # Adjust the interval as needed

except KeyboardInterrupt:

pass

finally:

conn.close()

GPIO.cleanup()

Creating a complete app that receives and displays parking availability data from a Raspberry Pi using Python and the Flutter framework is a complex project that involves both frontend and backend development. I can provide you with a basic outline of the steps involved and some code snippets to get you started:

**1. Set up the Raspberry Pi:**

- Connect sensors or cameras to the Raspberry Pi to collect parking availability data.

- Use Python on the Raspberry Pi to process this data and expose it through an API.

**2. Create a Flutter App:**

- Set up a Flutter development environment.

- Create a new Flutter project.

- Add necessary dependencies for making HTTP requests and building the UI.

**3. Implement Flutter UI:**

- Create a user interface to display parking availability data. This might include a map, list, or any suitable visualization.

- Design the UI to show the data dynamically as it's received.

**4. Create API Requests in Flutter:**

- Use Flutter's HTTP package (such as `http` or `dio`) to make API requests to the Raspberry Pi.

- Retrieve parking availability data from the Raspberry Pi using these requests.

**5. Update UI with Real-time Data:**

- Implement a mechanism to continuously fetch and update parking availability data from the Raspberry Pi.

- Update the UI to reflect the real-time data.

**Here's a simplified Python script to serve as a starting point on the Raspberry Pi:**

```python

# Import necessary libraries (e.g., Flask for creating a web API)

from flask import Flask, jsonify

app = Flask(\_\_name)

# Simulated parking availability data

parking\_data = {

'available\_spots': 50,

'total\_spots': 100,

}

@app.route('/get\_parking\_data', methods=['GET'])

def get\_parking\_data():

return jsonify(parking\_data)

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=5000)

```

In your Flutter app, you can make HTTP requests to the Raspberry Pi's endpoint (e.g., `http://raspberry\_pi\_ip:5000/get\_parking\_data`) to retrieve parking availability data. You'll need to parse the JSON response and update your app's UI accordingly.

**Code Implementation:**

**1: Sensor Integration with Arduino**

This module involves integrating the IR sensor with the Arduino to detect the presence or absence of cars in parking spaces.

IR Sensor Data Processing Algorithm (Module 1):Simple threshold-based algorithm to detect the presence or absence of cars based on sensor readings.Display Logic

Code:

Arduino:

#include <NewPing.h>

#define TRIGGER\_PIN 9

#define ECHO\_PIN 10

#define MAX\_DISTANCE 200

#define NUM\_PARKING\_SPOTS 4

#define THRESHOLD\_DISTANCE 20 // Adjust this threshold distance as needed

NewPing sonar(TRIGGER\_PIN, ECHO\_PIN, MAX\_DISTANCE);

int parkingSpots[NUM\_PARKING\_SPOTS] = {0}; // Array to store parking spot statuses

void setup() {

Serial.begin(9600);

}

void loop() {

// Check each parking spot

for (int spot = 0; spot < NUM\_PARKING\_SPOTS; spot++) {

int distance = sonar.ping\_cm();

if (distance < THRESHOLD\_DISTANCE) {

// Vehicle is detected in the parking spot

if (parkingSpots[spot] == 0) {

Serial.print("Parking Spot ");

Serial.print(spot + 1);

Serial.println(" occupied.");

}

parkingSpots[spot] = 1; // Set the spot as occupied

} else {

// No vehicle in the parking spot

if (parkingSpots[spot] == 1) {

Serial.print("Parking Spot ");

Serial.print(spot + 1);

Serial.println(" vacant.");

}

parkingSpots[spot] = 0; // Set the spot as vacant

}

delay(100); // Delay to prevent rapid sensor readings

}

}

**2:Display Interface with Arduino:**

This module focuses on setting up the LCD display with the Arduino to provide real-time parking space information to users.

Servo Motor Control Algorithm/ Display control algorithms :Code to update the LCD display with real-time parking space information, which could involve basic text rendering and update routines.

Code:

#include <Adafruit\_GFX.h>

#include <Adafruit\_NeoMatrix.h>

#include <Adafruit\_NeoPixel.h>

#define MATRIX\_PIN 6 // Define the pin for the LED matrix

#define NUM\_ROWS 4

#define NUM\_COLS 4

Adafruit\_NeoMatrix matrix = Adafruit\_NeoMatrix(NUM\_COLS, NUM\_ROWS, MATRIX\_PIN,

NEO\_MATRIX\_TOP + NEO\_MATRIX\_LEFT +

NEO\_MATRIX\_ROWS + NEO\_MATRIX\_PROGRESSIVE,

NEO\_GRB + NEO\_KHZ800);

int parkingSpots[NUM\_ROWS \* NUM\_COLS] = {0}; // Array to store parking spot statuses (0 for vacant, 1 for occupied)

void setup() {

matrix.begin();

matrix.setTextWrap(false);

matrix.setBrightness(50); // Adjust the brightness as needed

matrix.setTextColor(NEO\_GRB + NEO\_KHZ800);

}

void loop() {

// Check the status of each parking spot

for (int spot = 0; spot < NUM\_ROWS \* NUM\_COLS; spot++) {

if (parkingSpots[spot] == 0) {

// Spot is vacant, set LED to green

matrix.drawPixel(spot % NUM\_COLS, spot / NUM\_COLS, matrix.Color(0, 255, 0));

} else {

// Spot is occupied, set LED to red

matrix.drawPixel(spot % NUM\_COLS, spot / NUM\_COLS, matrix.Color(255, 0, 0));

}

}

matrix.show();

// You can replace this with your sensor code to update parking status

// Example: parkingSpots[0] = digitalRead(sensorPin);

delay(1000); // Update the display every second or adjust as needed

}

**3: Servo Motor Control with Arduino**

This module involves using the servo motor with the Arduino to physically indicate the status of a parking space (occupied or vacant).

PWM (Pulse Width Modulation) control for servo motor.

Code:

#include <Servo.h>

Servo parkingGateServo;

int servoPin = 9; // The PWM pin to which the servo is connected

int closedPosition = 0; // Servo position for the gate closed

int openPosition = 90; // Servo position for the gate open

int delayTime = 2000; // Time to keep the gate open (in milliseconds)

void setup() {

parkingGateServo.attach(servoPin);

parkingGateServo.write(closedPosition); // Initialize the gate as closed

delay(1000); // Wait for the servo to reach the closed position

}

void loop() {

// Check parking spot availability and control the servo accordingly

int isParkingAvailable = checkParkingAvailability(); // Replace with your parking spot sensor logic

if (isParkingAvailable) {

openGate();

delay(delayTime);

closeGate();

}

}

void openGate() {

parkingGateServo.write(openPosition);

delay(1000); // Wait for the gate to open

}

void closeGate() {

parkingGateServo.write(closedPosition);

delay(1000); // Wait for the gate to close

}

int checkParkingAvailability() {

// Implement your parking spot sensor logic here

// Return 1 for available, 0 for not available

// Example: Replace with actual sensor code

// int sensorValue = digitalRead(sensorPin);

// if (sensorValue == HIGH) {

// return 0; // Parking spot is occupied

// } else {

// return 1; // Parking spot is available

// }

return 1; // Placeholder, always returns available

}

**4.User Interface (Mobile App/Web App) Development:**

This module involves developing a mobile app or web app that displays parking space availability information to users.

UI/UX design algorithms, data retrieval and display logic, possibly API integration for communication with the central server or cloud platform.

Code:

<!DOCTYPE html>

<html>

<head>

<title>Smart Parking System</title>

<link rel="stylesheet" type="text/css" href="style.css">

</head>

<body>

<h1>Smart Parking System</h1>

<div class="parking-spot" id="spot-1">

<h2>Parking Spot 1</h2>

<p>Status: <span id="status-1">Unknown</span></p>

</div>

<div class="parking-spot" id="spot-2">

<h2>Parking Spot 2</h2>

<p>Status: <span id="status-2">Unknown</span></p>

</div>

<!-- Add more parking spots as needed -->

<script src="script.js"></script>

</body>

</html>

css:

body {

font-family: Arial, sans-serif;

text-align: center;

margin: 20px;

}

h1 {

color: #333;

}

.parking-spot {

background-color: #f0f0f0;

border: 1px solid #ccc;

padding: 20px;

margin: 10px;

}

.parking-spot h2 {

font-size: 20px;

color: #555;s

}

.parking-spot p {

font-size: 16px;

color: #777;

}

Script.js:

// Simulated parking spot status

const parkingStatus = {

spot1: 'Vacant',

spot2: 'Occupied',

// Add more spots and statuses as needed

};

// Function to update the UI with the current status

function updateUI() {

Object.keys(parkingStatus).forEach((spot) => {

const statusElement = document.getElementById(`status-${spot}`);

statusElement.textContent = parkingStatus[spot];

});

}

// Simulate data updates (you should replace this with real data from your smart parking system)

setInterval(() => {

parkingStatus.spot1 = Math.random() < 0.5 ? 'Vacant' : 'Occupied';

parkingStatus.spot2 = Math.random() < 0.5 ? 'Vacant' : 'Occupied';

// Update other spots as needed

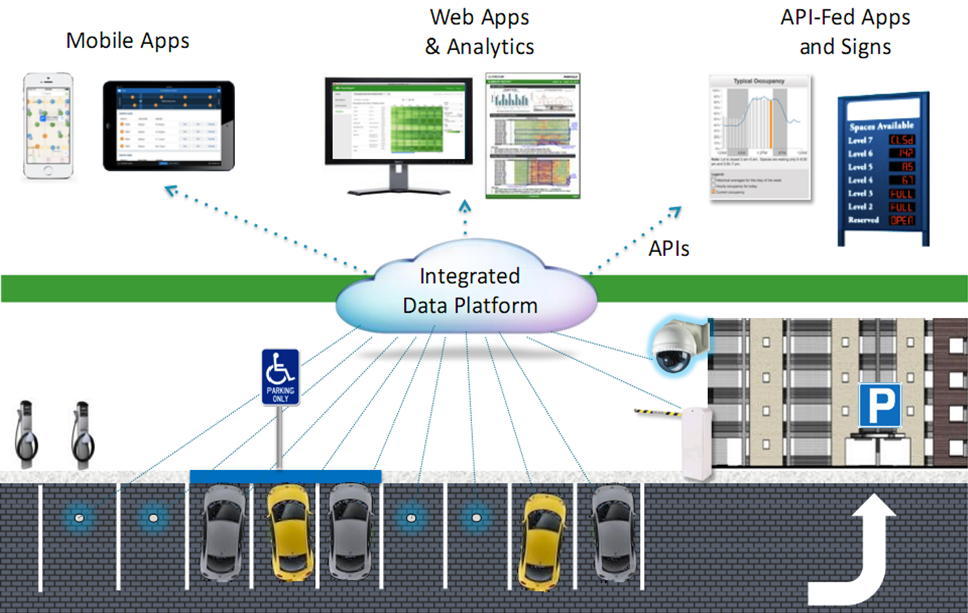
updateUI();

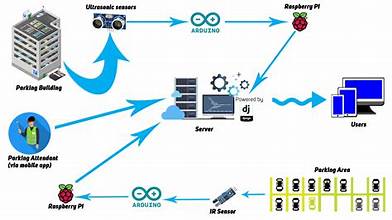
}, 5000); // Update every 5 seconds

// Initial UI update

updateUI();

**Example Diagram:**





**Raspberry pi Data Storage Output Structure:**

The provided Python script is for reading data from a parking spot sensor connected to a Raspberry Pi and storing the data in a SQLite database. The script doesn't actually provide an "output" in the traditional sense but is meant to continuously monitor the sensor's status and store it in the database. However, here is a simplified explanation of how the script works and what you can expect when you run it:

1. **Initialization**:
   * The script initializes the Raspberry Pi's GPIO pins using the **RPi.GPIO** library and specifies the GPIO pin (in this case, pin 17) connected to the sensor.
2. **Database Initialization**:
   * It connects to an SQLite database named "smart\_parking.db" and creates a table called "parking\_data" if it doesn't already exist. This table is used to store the status of the parking spot (occupied or not) and a timestamp.
3. **Sensor Reading Loop**:
   * Inside the **while True** loop, it continuously reads data from the sensor using **GPIO.input(sensor\_pin)**.
4. **Data Storage**:
   * It then inserts the sensor's data (0 for unoccupied and 1 for occupied) into the "parking\_data" table of the SQLite database along with a timestamp.
5. **Debug Output**:
   * For debugging purposes, it prints the sensor data (0 or 1) to the console at each iteration. This output is not typically needed in a production system.
6. **Sleep**:
   * After each sensor reading and data insertion, the script pauses for 5 seconds (adjustable) before reading the sensor again. You can modify this sleep duration to control how often the sensor is polled.
7. **Keyboard Interrupt**:
   * The script can be terminated by pressing Ctrl+C, at which point it will perform cleanup actions, closing the database connection and releasing the GPIO pins.

To run this script, ensure that you have the **RPi.GPIO** library installed on your Raspberry Pi and a sensor connected to the specified GPIO pin. You should also have the **sqlite3** library available. You can use SQLite database browser tools to view and analyze the stored data in the "smart\_parking.db" database.

**Application:**

1. **Real-Time Parking Availability Information:** IoT sensors, such as ultrasonic distance sensors or cameras, are deployed in parking spaces to monitor occupancy. This data is transmitted to a central server, and through a mobile app or other means, drivers can access real-time information about available parking spots in the vicinity. This allows drivers to make informed decisions about where to park, reducing the time spent searching for a spot.
2. **Reduced Congestion and Pollution:** By guiding drivers to available parking spaces efficiently, Smart Parking systems help reduce traffic congestion and the associated greenhouse gas emissions. Drivers spend less time circling the block searching for parking, which contributes to a decrease in pollution and fuel consumption.
3. **Enhanced User Experience:** Drivers benefit from a more convenient and stress-free parking experience. They can find and reserve parking spaces before arriving at their destination, making it easier to plan their trips.
4. **Optimized Space Utilization:** Smart Parking systems help parking lot operators and city planners make more effective use of available parking spaces. The data collected from IoT sensors can be analyzed to identify usage patterns and trends. This information can guide decisions on optimizing parking facilities, pricing strategies, and infrastructure development.
5. **Revenue Generation:** Parking operators can implement dynamic pricing based on real-time demand. As parking spaces become scarcer, the price can increase, generating additional revenue. This dynamic pricing model helps balance supply and demand.
6. **Security and Safety:** Some Smart Parking systems incorporate security features such as surveillance cameras or panic buttons for safety concerns. This enhances the overall safety and security of parking facilities.
7. **Remote Monitoring and Maintenance:** IoT sensors can provide real-time data on the condition of parking infrastructure. This allows for proactive maintenance and rapid response to issues like malfunctioning payment machines or security breaches.
8. **Integration with Navigation Apps:** Smart Parking systems can integrate with popular navigation and mapping apps. Drivers can receive parking availability information and navigation guidance seamlessly, making it even more convenient to find parking.
9. **Environmental Benefits:** Reduced congestion and more efficient parking contribute to lower carbon emissions, helping to combat air pollution and support sustainability goals in urban areas.
10. **Data-Driven Insights:** The data collected by IoT sensors can be analyzed to gain insights into parking behavior, which can be valuable for urban planning and traffic management. This data can inform decisions on expanding parking facilities, implementing smart city initiatives, and improving transportation systems.
11. **Reduced Frustration and Stress:** Searching for parking can be a significant source of frustration for drivers. Smart Parking systems alleviate this stress, making urban travel more pleasant.
12. **Accessibility and Inclusivity:** Smart Parking systems can be designed to include features that aid people with disabilities. These features may include designated accessible parking spots and real-time availability information for these spaces.

**Advantages**

The importance of smart parking is:

• Accurately sense and predict spot/vehicle occupancy in real-time.

• Guides residents and visitors to available parking spot.

• Optimize Parking Space Usage.

• Simplifies the parking experience and adds value for parking stakeholders, such as merchants and drivers.

• Helps the free flow of traffic in the city leveraging IoT technology.

• Enables intelligent decisions using data, including real– time status applications and historical analytics reports.

• Smart Parking plays an important role in creating better urban environment by reducing the emission of CO2 and other pollutants

**CONCLUSION:**

Smart Parking systems powered by IoT represent a pivotal solution to the pervasive challenges of urban parking. By providing real-time parking availability information to drivers, these systems facilitate more efficient utilization of parking spaces, thereby reducing traffic congestion, emissions, and driver frustration. The integration of mobile apps and IoT sensors enhances the user experience, making it easier for individuals to find parking quickly and seamlessly. Additionally, the data generated by these systems offers valuable insights for urban planners and parking operators, allowing them to optimize resource allocation and support dynamic pricing models. Furthermore, the security and safety features, environmental benefits, and accessibility considerations make Smart Parking through IoT a multifaceted solution that contributes to more sustainable and user-friendly urban environments. As urban populations continue to grow, such innovations play a crucial role in addressing parking issues and improving the overall quality of urban life.